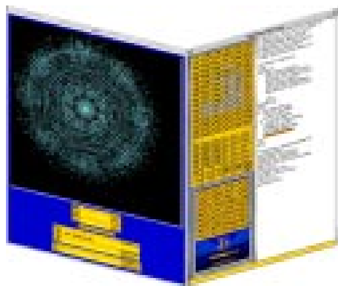
CODEMILL-J 4.0 USER GUIDE



SABATECH CORPORATION

2004

Copyright © 1999-2004. Sabatech Corporation. All rights reserved. Sabatech Corporation have made every effort to make this guide and accompanying computer application, CodeMill-J free of errors and defects. It, however, offers no warranty of any kind, expressed or implied with regards to the information contained in the guide and concepts demonstrated by the application. They shall not be liable in any event for incidental and consequential damages in connection with the use of the guide and CodeMill-J.

Table of Contents:

1. Launching CodeMill-J
    1.1. Running your first program
    1.2. Sample Programs

2. Setting Options
    2.1. Setting Scratchpad and Register Size
    2.2. Setting RAM, VRAM, and STACK Size
    2.3. Setting Monitor Options
    2.4. Setting the Arithmetic Mode

3. Viewing Selections
    3.1. Clearing Registers
    3.2. Clearing and Hiding Scratchpad registers
    3.3. Clearing Screen and Hiding Mesh

4. File and Edit Menus
    4.1. File Menu
    4.2. Edit Menu

5. Running Programs
    5.1. Entering Programs
    5.2. Running Programs
    5.3. Structure of CodeMill-J Programs
    5.4. Error Messages

6. A Sample Session in CodeMill-J

7. Assembly Language Programming in CodeMill-J
    7.1. Computing Formulas in CodeMill-J
    7.2 Entering and Displaying Formulas in CodeMill-J
    7.3. Commenting in CodeMill-J

8. CodeMill-J Instruction Set and Commands
    8.1. CodeMill –J Instruction Set
    8.2 CodeMill Commands

9. CodeMill-J Help on the Web

1. Launching CodeMill-J

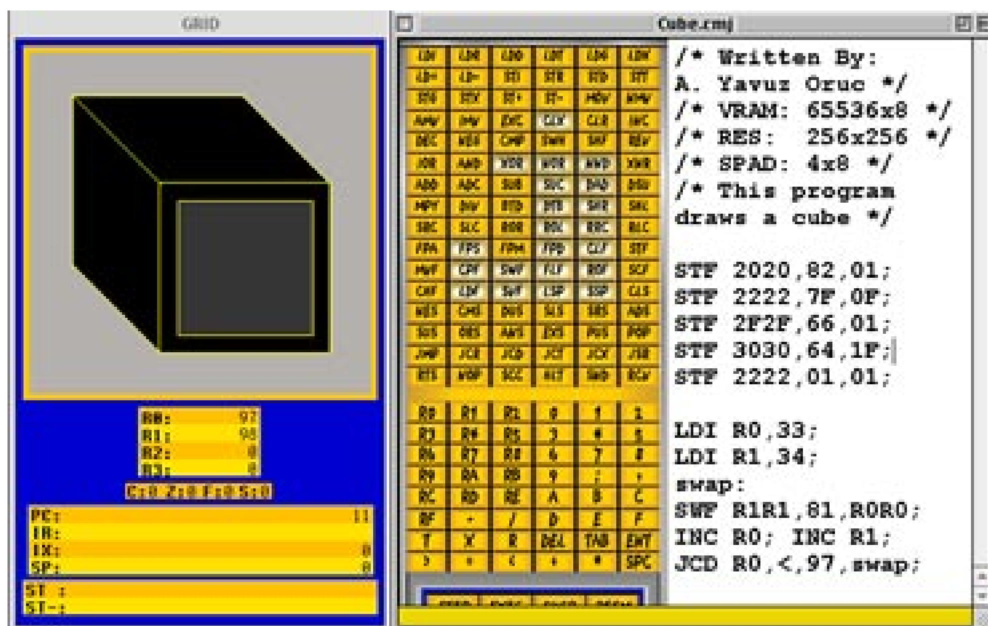
As with other applications, you bring up CodeMill-J by double clicking the CodeMill-J icon in the CodeMill-J folder or selecting it from the applications menu on your computer. Once it is launched, CodeMill-J is up and running, and you can enter your program in the program window (Console) on the right, and press the **STEP** or **EXEC** button on the CodeMill-J palette when you wish to step through or execute it. Unlike other assemblers, running CodeMill-J does not require any directives to run a program. Just enter your program, and press the **STEP** or **EXEC** button. It is that simple!

1.1. Running Your First Program

What good is an application if you cannot get started with it right away? So, if you want to see how CodeMill-J works, the simplest thing to do on it is to set a pixel on the screen. Just type in **STF 0808, 01,03;** and press execute. That is not too impressive, but it gives you an idea how you can compose and run programs on CodeMill-J. Try to guess what each piece of information in the code does. For example, try typing and executing **STF 0507,02,04;** to see what happens.

1.2. Sample Programs

CodeMill-J CD comes with more than 200 programs that have been written by CodeMill-J experts and enthusiasts. If you would like to try some of these programs, browse the sample programs folder, and launch and execute them in CodeMill-J. For example, the image below shows how to draw a cube in CodeMill-J. You will soon be writing programs that are as neat as those programs yourself :-).



2. Setting Options

CodeMill-J has a number of options which you can set any time during a CodeMill-J session, even as you are running programs (see below).

Options Help
Scratchpad Size
RAM Size
VRAM Size
STACK Size
Monitor Resolution
Monitor Size
Screen Color
Screen Scene
Arithmetic
Font
Style
Size
Soft Warnings

The options determine how CodeMill-J is configured for executing programs and displaying its behavior during the execution of programs. For example, you can change the resolution of the GRID screen to make the graphics more realistic using the **Monitor Resolution** option, or you can allocate more registers to make sure that your program can run more efficiently and faster using the Scratchpad Size option. You can also change the options using the command buttons on the CodeMill-J palette as we will see in the next section. *(Note: Some options shown here may not be available on your version of CodeMill-J.)*

2.1. Setting Scratchpad and Register Size

The number of registers in CodeMill-J's scratchpad, and their size are set using the **Scratchpad Size** option from the **Options menu**. This option is designed to simulate computers with different register sizes, and different number of registers. To set the scratchpad size, use the **Scratchpad Size submenu** in the **Options menu**. The available sizes are

4 x 8, 8 x 8, 16 x 8, 4 x 16, 8 x 16, 16 x 16, 4 x 32, 8 x 32, and 16 x 32,

where the first value specifies the number of registers, and the the second value specifies the size of each register in the scratchpad. The number of bits

in a register determines the range of numbers that can be represented by CodeMill-J. An 8-bit register can hold any integer between –128 and 127 whereas a 16-bit register can hold any integer between –32768 and 32767. The umber of registers bounds the number of variables in a computer program that can be mapped to the scratchpad of CodeMill-J without having to save the registers to the memory during computations. Registers are fast memory locations that reside inside the central processor units of computers. Since the space is limited inside a central processor unit, today's computers come with only a small number, typically, 16 to 32 registers. This can, of course, change as computers are designed and implemented with smaller electronic cirtcuits.

You can also use the SPSZ+ and SPSZ- buttons on the CodeMill-J palette to choose the next or previous scratchpad size option with respect to the currently selected scratchpad size option (see below).

2.2. Setting RAM, VRAM, and STACK Size

The RAM, VRAM, and STACK sizes can be set using the corresponding submenus in the **Options menu**. This option is designed to simulate computers with different word size and RAM, VRAM and STACK memory capacities. The RAM memory typically holds operands that can not be kept in the scratchpad because of its limited capacity. The VRAM holds the pixels on CodeMill-J's GRID screen. The STACK is used to run certain types of programs and save the return address in subroutine calls. In all three memory settings, the available choices depend on the type of memory that is being set. The first value specifies the number of locations, and the second value specifies the number of bits in each location. For example, if you set the RAM size to 1024 x 16, CodeMill-J assumes that you are using a RAM memory with 1024 locations, each holding 16 bits. VRAM and STACK memory sizes are similarly set.

You can also use the RMSZ+ and RMSZ- buttons on the CodeMill-J palette to choose the next or previous RAM size option with respect to the currently

selected RAM size option (see below). Similarly, VRAM and STACK sizes can be adjusted from the CodeMill-J palette using the VRSZ+, VRSZ-, STSZ+, and STSZ- buttons.

STEP	EXEC	SUSP	RESM
RAM	VRAM	STCK	CPUR
RSTA	CSRC	RSPC	RSSP
RSIX	H/SCN	H/SRN	H/SPD
BNRY	OCTL	DCML	HXDM
SPSZ+	SPSZ-	RMSZ+	RMSZ-
VRSZ+	VRSZ-	STSZ+	STSZ-
RESL+	RESL-	SCRN+	SCRN-
SCN+	SCN-	COLR+	COLR-

2.3. Setting Monitor Options

The CodeMill-J's Monitor settings can be changed to adjust its size, resolution, background color, and background scene.

Resolution

The **resolution** of the monitor refers to the number of pixels along its height and width. CodeMill-J's screen is square-shaped so that it has the same number of pixels along its height and width. To set the resolution of CodeMill-J's screen, select the **Monitor Resolution submenu** from the **Options menu**, and click on the desired resolution. The available resolutions are

8x8, 16x16, 32x32, 64x64, 128x128, 256x256, 512x512.

In the lowest resolution, CodeMill-J displays 8 pixels along its height and width, providing 64 pixels in all. At the highest resolution, it displays 512 pixels along its height and width, providing 512x512 = 262,144 pixels

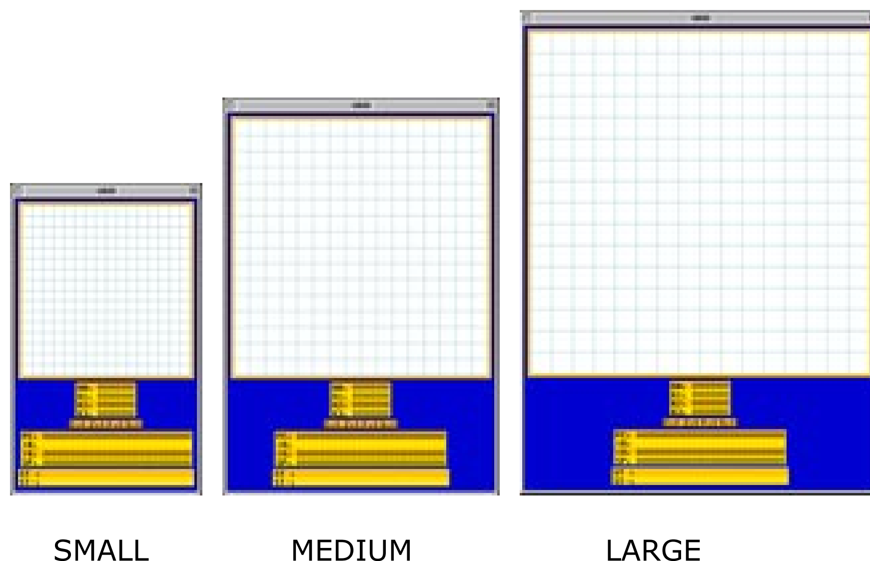
You can also use the RESL+ and RESL- buttons on the CodeMill-J palette to choose the next or previous resolution option with respect to the currently selected resolution size option.

Screen Size

The **screen size** of CodeMill-J can also be changed. CodeMill-J comes with three screen sizes as shown below at 25% of the full scale, called *small,*

medium, and large. To set the **screen size**, choose the **Monitor Size submenu** from the **Options menu**, and click on the size you desire.

Note: The 512x512 resolution can only be used when the screen size is set to large as the actual (physical) sizes of the small or medium screens are smaller than 512 pixels. If you set the resolution to 512x512 with the small or medium size screen, CodeMill-J does not display any pixels.



SMALL MEDIUM LARGE

Background Color

The **background color** of CodeMill-J's screen can be set to a variety of colors that include the following:

white, black, gray, orange, red, green, blue, yellow.

To set the background color of CodeMill-J's screen, choose the **Screen Color submenu** from the **Options menu**, and click on the color you desire.

You can also use the COLR+ and COLR- buttons on the CodeMill-J palette to choose the next or previous resolution option with respect to the currently selected color option.

Background Scene

The **background scene** of CodeMill-J's screen can be set to a variety of background scenes that are listed under the background scene menu or by

opening a background scene from a file. The image below depicts the "Houses I" background scene.

These options are designed to visualize the effects of various resolution sizes on the quality of pictures and images and enhance animation features of CodeMill-J.

2.4. Setting the Arithmetic Mode

The contents of registers in CodeMill-J can be displayed in *binary, octal, decimal, or hexadecimal* number representations. These arithmetic display options are designed to examine various representations of numbers and operands in computers. Binary numbers use digits 0 and 1 only. Octal numbers use the digits 0,1,2,...,7. Decimal numbers use the digits 0,1,2,...,9. Hexadecimal numbers use 0,1,2,...,9,A,B,C,D,E,F. All these four number systems use positional number representation to determine the value of a number with a **radix**. For example,

101 in binary = 1x4 + 0x2 + 1x1 = 5 in decimal
377 in octal = 3x64 + 7x8 + 7 = 255 in decimal.
FFF in hexadecimal = 15x256 + 15 +16 + 15 = 4095 in decimal.

To set the arithmetic mode select the **Arithmetic Mode submenu** from the **Options menu**, and click on the arithmetic mode you desire. In addition to these four modes, CodeMill-J can also display the register contents in floating-point number format. However, displaying register contents in this format makes sense only if you use CodeMill-J's floating-point operations.

3. Viewing Selections

In addition to the **Options menu**, CodeMill-J has a menu, called View (see below), for setting, clearing the contents of CodeMill-J's registers and screen, hiding the register set,the mesh and background scene on the screen. These options are designed to reset registers and eliminate the left over images on

the screen before a new program is run. In particular, hiding the register set boosts the speed of drawing pictures on CodeMill-J's screen.

View Options Help

Show GRID Settings
Hide GRID Window

Show Scene ⌘/
Hide Mesh ⌘E
Clear Screen ⌘F

Hide CPU Registers ⌘G
Clear Scratchpad ⌘J
Reset PC ⌘K
Reset IX
Reset SP
Reset Flags
Reset All ⌘L

Hide Palette ⌘\

3.1. Clearing Registers

The PC, IX, and SP registers, and the flags can be cleared by choosing the **Reset PC**, **Reset IX**, **Reset SP**, **Reset Flags** items from the **View menu**. The **Reset All** option clears all the registers and flags.

3.2. Clearing and Hiding Scratchpad Registers

The scratchpad registers can be cleared by choosing the **Clear Scratchpad** item from the **View Menu**. The registers can be hidden from view by choosing the **Hide Scratchpad** item from the same menu. It can be brought back to view by choosing the **Show Scratchpad** item.

3.3. Clearing Screen and Hiding Mesh and Background Scene

The pixels on the screen can be cleared by choosing the **Clear Screen** item from the **View Menu**. The mesh on the screen can be hidden from view by choosing the **Hide Mesh** item from the **View Menu**, and it can be brought back to view by choosing the **Show Mesh** item. Likewise, the background scene can be brought to view using the **Show Scene** option or hidden from view using the **Hide Scene** option.

3.3. Hiding GRID Window

The GRID window can be hidden from view by choosing the **Hide GRID Window** from the **View Menu**. It can be brought back to view by choosing the **Show GRID Window** from the same menu. The same effect can be achieved by double clicking on the menu bar of GRID window if MacOs 8.0 or higher has been installed on your computer.

3.3. Viewing CodeMill-J Settings

The current option settings of CodeMill-J can be viewed by choosing the **Show Grid Settings** item from the **View menu**. This option is designed to provide a point of access to all the option settings of CodeMill-J.

4. File and Edit Menus

File and **Edit** menus work like the usual File and Edit menus for most applications with some minor differences.

4.1 File Menu

The file menu has the usual **New**, **Open**, **Close**, **Save**, and **Save As** options for opening a new Program Window, opening or closing an existing program, or saving a program. The **Page Set up**, **Print One**, and **Print** options are used to print the **Program** or **Grid Window** whichever is active at the time the **Print submenu** is selected.

4.2 Edit Menu

First, the **Edit Menu** works only with the **Console** or **Program Window** on the right. The **Cut**, **Paste**, **Copy**, **Clear**, **Undo** options are used to insert, replicate and delete portions of a program in the Console.

5. Program Entry and Execution

5.1. Entering Programs

Programs can be keyed in using CodeMill-J's instruction palette, or entered more directly by typing them on the keyboard. In entering programs, it is important to insert space between different portions of a statement unless they are separated by a comma.

5.2. Running Programs

CodeMill-J programs are run using **STEP, EXEC, SUSP, and RESM** keys on the **Instruction Palette.** Stepping (STEP) is useful for debugging programs, while executing (EXEC) a program is used to watch the program's overall effect on CodeMill-J's registers and screen. You can suspend a

program any time you wish by pressing the **SUSP** key, and resume it by pressing the **RESM** key.

5.3. Structure of CodeMill-J Programs

Loosely speaking, a CodeMill-J program is a sequence of CodeMill-J instructions put together. There is no requirement that a CodeMill-J program begin with a specific preamble or a header, neither is there supposed to be a termination statement,
even though, a HLT instruction is often inserted at the end of a program to mark its end. Given this flexibility, CodeMill-J is empowered with the ability of executing programs on the fly. To witness this flexibility before you start writing your own programs, you may find it useful to try some of the programs in **Sample Programs Folder** . These programs also illustrate how various CodeMill-J Instructions work. To run any of these programs, first locate them in **Sample Programs Folder** using the **Open command** in the **File Menu**. Once a sample program is opened, it can be executed by pressing **EXEC** key on CodeMill-J's **Instruction palette**. Typically, each program contains a comment section at the top which specifies the default CodeMill-J options for it to run. Make sure that the current option settings match the settings in the comment section of the program you have selected to run.

5.4. Error Messages

CodeMill-J may issue an error message during the execution of a program. In general, an error message refers to either a syntax error in the program, or an incompatibility of an operation with the current option settings of CodeMill-J. The option settings are designed to expose the user to potential size and structure constraints between its various components. CodeMill-J stops executing when it encounters an error condition. If the error is caused by an invalid syntax, the statement which contains the error is found in the IR register. If it is caused by an incompatible setting, the error message typically reveals which option setting may have caused the error. Just edit your program and/or change your option settings to fix problem, and press **EXEC** or **RESM** key to resume execution.

6. A Sample Session in CodeMill-J

(1) Bring up CodeMill-J as described earlier. (Double click on its icon, or open it from the file or start menu).

(2) The cursor must be blinking on the top left hand corner of the text area in the Console window. This means that you can enter your program starting in that position. However, before you do so, it is important to check the settings on the GRID window and choose the right options for the screen and scratchpad registers.

(3) The GRID window on the left has been initialized by default to an 8 X 8 screen. This means that CodeMill-J's screen contains eight pixels in each row and in each column. Also, the scratchpad has been initialized to four registers, each holding 16 bits. These settings can be changed simply by first clicking on the GRID window, then pressing on the Options menu, and then selecting the desired option as explained in the earlier section. Select the $8 X 8 option from the **Scratchpad submenu**. You will see that the scratchpad now contain 8 registers, and each register holds 8 bits.

(4) Set the screen resolution to 16 x 16. You will get an error message indicating that you have insufficient VRAM. This is because a 16 X 16 screen needs 256 bytes of VRAM at one byte per each pixel, but the default is set to 64 bytes. So change the setting for VRAM to allocate at least 256 bytes or any number higher than hat in the VRAM menu, and then reset the resolution to 16 x 16. You should see a screen with 256 pixels on it. As noted before, CodeMill-J's screen can be set to have as many as 256 pixels in each row and column.

(5) Switch to the Console window, and type **STF 0000,04,03;** (Make sure that you type STF in capital letters.) As an alternative you can use the **Instruction Palette** to enter the same line by pressing the following sequence of buttons:

STF, space, zero zero zero zero, zero four, zero, three;

(6) Now move the cursor over to the **STEP key**, and click the mouse while watching the window on the left. You will see a red square come up on the left hand corner. The square spans 4 pixels in the horizontal and vertical directions. Why is it red? Well, the last two digits, 03, is the hexadecimal code for color red. Why is it located at the top left hand corner? Because, the first two zeros in the sequence 0000 identify the horizontal position of the square relative to the top left hand corner, and the second two zeros identify the vertical position of the same square relative to the same corner. The prefix STF is the **mnemonic (opcode)** for the **SeT Frame instruction** that creates a **square frame** whose top left corner is located at the hexadecimal address as specified in the program. The hexadecimal code makes coding the addresses of pixels easy. The pixels are numbered in 2-digit hexadecimal notation from left to right and top to bottom, starting with 00 along each axis.

(7) Now, enter **STF 0C0C,04,01;** in the next line in the Console window. (Use the return key on the keyboard or the **ent** button on the palette to go to the next line). If you again press the STEP key, you will see a yellow square with the same dimensions, but this time, it is at the bottom right corner. The reason is, of course, the address and color have both changed while the size of the frame has remained fixed at 4.

(8) Now enter **SWF 0000,04,0C0C;** in the next line in the Console window, and press the **STEP key**, while watching the screen on the left. You will notice that the colors of the pixels in the two frames have changed to opposite colors as shown in Figure \ref{f23}. Well, SWF is the **SWap Frames** i**nstruction**, and 0000, and 0C0C in the instruction specify the addresses of the frames to be swapped, while the infix (middle operand), 04 fixes the size of the frames in the swap.

(9) Now type JMP D,2; in the next line in the Console window, and this time move the cursor to the **EXEC key**, and press it. You will see on the left that the frames are flashing with alternating colors. What is happening is that you've put in place a program loop within which you are constantly swapping the two square frames, and hence the frames appear to have their colors alternate. **JMP** is the **jump instruction**, D signifies a form of addressing, called **direct addressing**, and 2 is the address of the instruction the program jumps back to. You notice how the instructions are moving through the instruction register (IR). That is where the instructions go to before they are executed. Also notice how the bits in the program counter (PC) are changing.

(10) Now while your code is executing, select Hide CPU registers from the View menu, and notice that the code is executing much faster.This is because, CodeMill-J is relieved from updating the registers in this case, and this saves time.

(11) If you are thinking about how to stop CodeMill-J from executing your code forever, just press on **SUSP key** while your code is executing; You should notice that the program stops executing, and if you press **RESM**, it begins to execute again. So, you can suspend and resume your code as many times as you wish. You can also step it after you suspended it.

(12) Finally, to quit CodeMill-J, press on the **File menu** , and choose **Quit**. Like any other application, CodeMill-J will quit execution, but before quitting, it will ask you if you wish to save the program you have entered. If you feel like saving the program you entered, select the {Save} command, and enter a file name (any name you like), and press on the **Save item** again. If not, press on the **Discard item**, or you wish to return to the program, press **Cancel**. You can launch CodeMill-J any time either by double clicking on its icon, or using the File menu.

7. Assembly Language Programming in CodeMill-J

CodeMill-J assembly language consists of

* 96 instruction keywords,
* 16 register specifiers, R0,R1,R2,...,R15,
* 16 numerical characters, 0,1,...,9,A,B,C,D,E,F,
* 9 relational operators, =,<=,>=,>,<,<>,<+,>+,!=
* 3 numerical specifiers, X,T,-
* 4 address mode specifiers, D,T,R,X
* a field separator, ``,'' and a statement separator, ``;''

The instruction keywords define the operations that can be performed by the CodeMill architecture on which CodeMill-J programs are run. The instruction palette in the Console window contains all these 96 instructions. They can be entered into CodeMill-J programs either by pressing on the key on the instruction the palette, or directly by using the keyboard of your computer. Each instruction has a format or syntax that defines the number of operands, and the role that each operand plays in the execution of that instruction. For example, **LDI** (derived from ``LoaD Immediate'') instruction requires one register, and one numerical operand. The numerical operand can be entered in octal, decimal, or hexadecimal notations. The decimal numbers are specified without any explicit specifier while the octal and hexadecimal numbers are specified using the specifiers T and X, respectively. Here are three different ways to load number 14 into register R1 using the LDI instruction:

LDI R1,14; LDI R1,T16; LDI R1,XE;

In each case the operand represents decimal 14. Negative numbers are entered in the same way in the three number systems by inserting "-" before the numerical operand, as in the following examples:

LDI R1,-14; LDI R1,T-16; LDI R1,X-E;

Each instruction and its operand fields collectively form a statement in CodeMill-J. As seen in the above examples, the statements in CodeMill-Junior are separated by ``;'' whereas the fields within a statement are separated by ``,''. Informally speaking, any sequence of syntactically correct statements constitutes a program in CodeMill-J though not all programs may produce meaningfull results. Here are two examples of CodeMill-J programs:

LDI R0,1; LDI R1,2; ADD R0,R1; HLT;

LDI R2,1; LDI R3,5; SUB R2,R3; HLT;

The first program loads 1 and 2 into registers R0 and R1,respectively, and then adds R1 to R0. The second programs loads 1 and 5 into R2 and R3, respectively, and subtracts R3 from R2. The **ADD** and **SUB** instruction both require two register operands, and the result of the operation is stored in the first register. In each program the last statement, called the HLT instruction forces the program to terminate.

Here are two other CodeMill-J programs:

LDI R1,X1F; LDI R2,12; INC R1; INC R2; ADD R2,R1; HLT;

LDI R2,T10; LDI R0,T-2;DEC R2; DEC R2; SUB R2,R0; HLT;

Can you guess what each program does? (Hint: **INC** increments its operand by 1 and **DEC** decrements its operand by 1). Can you further determine the values of the registers that appear in each program after it is executed?

7.1. Computing Formulas in CodeMill-J

Let us consider how a simple algebraic expression can be worked out in CodeMill-J. The expression is an ordinary sum of some four integers whose values may be varied without changing the program. This is facilitated in CodeMill-J as in other assembly language programs by referring to the operands in the summation through labels.

For example we may define the four operands as follows:

A: 10; B: 20; C: 30; D: 40;

where **A**, **B**, **C**, and **D** are labels, and **10**, **20**, **30**, and **40** are numerical values.

We can then load the four operands using four **LDI** instructions with labels **A**, **B**, **C**, and **D** as shown below:

LDI R0,A; LDI R1,B; LDI R2,C; LDI R3,D;

The only step that remains is to add the registers that can be done in three additions:

ADD R0,R1; ADD R0,R2; ADD R0,R3;

The entire program looks like as follows:

A: 10; B: 20; C: 30; D: 40;

LDI R0,A; LDI R1,B; LDI R2,C; LDI R3,D;

ADD R0,R1; ADD R0,R2; ADD R0,R3; HLT;

The sum **A + B + C + D** is accumulated in register **R0** after the program is executed. It should be noted that the values of **A**,**B**,**C**, and **D** can be changed to compute the sum of any four numbers as long as they are within the range of numbers representable in CodeMill-J. Also, the label statements need not appear before the instruction statements. In other words, the assignment of values to labels is location independent. In fact they can be moved anywhere in the program. For example, the following program should yield the same sum as the previous program.

LDI R0,A; LDI R1,B;

LDI R2,C; LDI R3,D;

ADD R0,R1; ADD R0,R2;

ADD R0,R3; HLT;

A: 10; B: 20;

C: 30; D: 40;

You can verify that both programs produce the same sum by rearranging the first program and executing it. Also you can use the STEP key to see the effect of each statement on the scratchpad registers during the execution of the program.

As a second example, consider the expression $1^2 + 2^2 + 3^2 + 4^2$.

This expression can be evaluated in several different ways two of which are given below:

Method 1:

Compute $1^2 = 1$, then compute $2^2 = 4$, then compute $3^2 = 9$, then compute $4^2 = 16$, then add all the squares, 1 + 4 + 9 + 16 = 30.

Method 2:

Compute $2^2 = 4$, then compute 1 + 4 = 5, then compute $3^2 = 9$, then compute 5 +9 = 14, then compute $4^2 = 16$, then compute 14+16 = 30.

Both methods produce the correct result, but the second one relies on less space from a computer's point of view. This is because the first computation needs to store the squares of all four numbers before they can be added. The second computation need only to have two locations, one to store the partial sum of the squares already computed, and another to store the square of the next number to be added to the partial sum. Moreover, storing the squares of the numbers requires that they are loaded back into the processor to compute their sum.

Ignoring the issue of space for the time being, and formalizing the second method, we are required to carry out the following computation: $((1 + 2^2) + 3^2)) + 4^2$.

As in the previous problem, we must assign one of the scratchpad registers to the result. Let us arbitrarily select R0. We can begin our computation by loading **1** into **R0**.

Next we must find a way to square **2** and add it to **R0**. We realize that another register must be used to compute the square of **2**. Let us arbitrarily select **R1** for this purpose. So our program up to now will look like this:

```
LDI R0,1; LDI R1,2;
```

The next problem is to compute 2^2. CodeMill-J has an instruction to multiply numbers, but this requires allocating two registers for the product that we do not want to deal with for the time being. Our only other option, is then to square our operands by successive addition. That is, we use the identities,

$2^2 = 2 + 2$, $3^2 = 3+3+3$, $4^2 = 4+4+4+4$.

We can compute 2^2 by adding **R1** to itself so that our program now should look like

```
LDI R0,1; LDI R1,2;

ADD R1,R1; ADD R0,R1; /* 1x1 + 2x2 is in R0 */
```

Repeating the last two steps for the other operands, the complete program should finally look like as shown below:

```
/* Load 1 */

LDI R0,1;

/* Compute 2x2 */

LDI R1,2; ADD R1,R1;
```

```
/* Compute 1x1 + 2x2 */

ADD R0,R1;

/* Compute 3x3 */

LDI R1,3; ADD R1,R1; LDI R2,3; ADD R1,R2;

/* Compute 1x1 + 2x2 + 3x3 */

ADD R0,R1;

/* Compute 4x4 */

LDI R1,4; ADD R1,R1; ADD R1,R1;

/* Compute 1x1 + 2x2 +3x3 + 4x4 */

ADD R0,R1;
```

In case you are wondering, expressions between **/*** and ***/** are comments, and are ignored by CodeMill-J when the program is executed.

Note that in computing **3+3+3**, we notice that we cannot obtain **9** by adding **R1** to itself twice. This will amount to computing **(R1 + R1) + R1**, and if we initially load **3** into **R1** as we do, at the end of the first sum, **R1** will then contain **6**. Now, adding **R1** to **R1** again will give **12**, not **9**. To get around this problem, we load **3** into **R2**, and add **R2** to **R1** after we add **R1** to itself.

As an alternative, we could add **3** to R0 after step **4**, and eliminate steps **7** and **8**. In either case, at the end of the 9th instruction, we should have **14** stored in **R0**. The last four instructions compute the square of **4**, add it to the partial sum to produce the result, and store it in **R0**.

LDI, ADD, and **SUB** are just three of several CodeMill-J's instructions with which we can carry out the arithmetic operations. Two other instructions that are closely related to **ADD** and **SUB** instructions are **INC** and **DEC**. The **INC** instruction increments a register by 1, whereas the **DEC** instruction decrements it by 1.

7.2 Entering and Displaying Formulas in CodeMill-J

CodeMill-J supports arithmetic in binary, octal, decimal and hexadecimal number systems. These number systems are described in great detail in *Introduction to Computers With CodeMill Junior*. Here we briefly mention how they are entered and displayed in CodeMill-J.

Decimal operands are entered into the operand fields of instructions directly, i.e., without using any identifiers.

Octal operands are prefixed by letter T, and the **hexadecimal operands** are prefixed by X.

For example,

LDI R0,T12;

loads the **octal 12** (**decimal 10**) into register **R0**. Likewise,

LDI R1,X1F;

loads **hexadecimal 1F** (**decimal 31**) into register **R1**. There is no specifier for binary operands in CodeMill-J. However, they can easily be specified using octal or hexadecimal specifiers. For example, the **binary number 01111011** (**hexadecimal 7B**) is loaded into register **R2** using the following statement

LDI R2,X7B; or LDI R2,T173;

To display the values of operands in CodeMill-J's scratchpad registers, you can use the **arithmetic submenu** in the **Options menu**. Simply click on the number representation in which you wish to display the operands. CodeMill-J will display them in the representation you selected.

7.3. Commenting in CodeMill-J

CodeMill-J programs can be commented by including them between pairs of /* and */.

For example,

/* This is a comment */
/*/* and so is this one */
/* but not this one.

8. CodeMill-J Instruction Set and Command Keys

CodeMill-J instruction set repertoire consists of the following 96 instructions and 36 commands.

8.1 CodeMill-J Instruction Set

LDI	LDR	LDD	LDT	LDG	LDX	LD+	LD-	STI	STR	STD	STT
STG	STX	ST+	ST-	MOV	NMV	AMV	IMV	EXC	CLX	CLR	INC
DEC	NEG	CMP	SWH	SHF	REV	IOR	AND	XOR	NOR	NND	XNR
ADD	ADC	SUB	SUC	DAD	DSU	MPY	DIV	BTD	DTB	SHR	SHL
SRC	SLC	ROR	ROL	RRC	RLC	FPA	FPS	FPM	FPD	CLF	STF
MVF	CPF	SWF	FLF	ROF	SCF	CHF	LDF	SVF	LSP	SSP	CLS
NES	CMS	DUS	SLS	SRS	ADS	SUS	ORS	ANS	EXS	PUS	POP
JMP	JCR	JCD	JCT	JCX	JSR	RTS	NOP	SCC	HLT	SND	RCV

/*LDI Help*/

Immediate Load instruction :

Syntax: ***LDI Rx, numerical or label operand;***

Examples:

LDI R1, 2;   /* loads 2 into R1. */
LDI R1, X20;   /* loads 32 into R1.*/
LDI R1, T20;   /* loads 16 into R1.*/
LDI R1, -2;   /* loads -2 into R1.*/
LDI R2, N; N:4; /* loads 4 into R2. */

/*LDR Help*/

Relative Load instruction:

Syntax: ***LDR Rx, numerical or label address;***

Examples:

LDR R1,20; /* loads the operand from memory location PC + 20 into R1.*/
LDR R2, N; N:10; /* loads the operand from memory location PC + 10 into R2. */

/*LDD Help*/

Direct Load instruction :

Syntax: *LDD Rx, numerical or label address;*

Examples:

LDD R1, 20; /* loads the operand in location 20 into R1.
LDD R2, N; N:10; loads the operand in location 10 into R2. */

/*LDT Help*/

Indirect Load instruction :

Syntax: *LDT Rx, numerical or label address;*

Examples:

LDT R1,20; /* loads the operand whose address is located in 20 into R1. */
LDT R2,N; N:10; /* loads the operand whose address is located in 10 into R2. */

/*LDG Help*/

Register Load instruction:

Syntax: *LDG Rx,Ry;*

Example:

LDG R1, R2; /* loads the operand whose address in memory is located in R2 into R1.*/

/*LDX Help*/

Indexed Load instruction:

Syntax: *LDX Rx, base,index;*

Example:

LDX R1,10,2; /* loads the operand whose address in memory is given by IX + 10 into R1. IX is replaced by IX + 2 after the load. */

/*LD+ Help*/

Autoincrement Load instruction:

Syntax: *LD+ Rx,Ry;*

Example:

LD+ R1,R2; /* loads the operand whose address is in R2 into R1, and then increments R2.*/

/*LD- Help*/

Autodecrement Load instruction :

Syntax: LD- Rx, numerical or label address;

Example:

LD- R1,R3; /* first decrements R3, and then loads the operand whose address is in R3 into R1. */

/*STI Help*/

Immediate Store instruction :

Syntax: STI numerical or label operand,numerical or label address;

Examples:

STI 10,20; /* stores 10 in location 20.*/
STI 5,N; N:10; /* stores 5 in location 10. */
STI N,20; N:5; /* stores 5 in location 20. */

/*STR Help*/

Relative Store instruction :

Syntax: STR Rx, numerical or label address;

Examples:

STR R1,20; /* stores R1 into location PC+20. */
STR R2,N; N:10; /* stores R2 into location PC+20.*/

/*STD Help*/

Direct Store instruction :

Syntax: STD Rx, numerical or label address;

Examples:

STD R1,X10; /* stores R1 into memory location 16. */
STD R1,N; N:T10; /* stores R1 into memory location 8. */

/*STT Help*/

Indirect Store instruction :

Syntax: ***STT Rx, numerical or label address;***

Examples:

STT R1,2; /* stores R1 into location whose address is found in location 2. */
STT R2,N;N:4; /* stores R2 into location whose address is found in location 4.*/

/*STG Help*/

Register Store instruction :

STG Rx,Ry;

Example:

STG R1,R2; /* stores R1 in location whose address is found in R2. */

/*STX Help*/

Indexed Store instruction:

Syntax: ***STX Rx,base,index;***

Example:

STX 10,2; /* stores R1 into memory location whose address is given by IX + 10 IX is replaced by IX + 2 after the store. */

/*ST+ Help*/

Autoincrement Store instruction :

Syntax: ***ST+ Rx,Ry;***

Example:

ST+ R1,R2; /* stores R1 in location whose address is found in R2. It then increments R2. */

/*ST- Help*/

Autodecrement Store instruction:

Syntax: ***ST- Rx,Ry;***

Example:

ST- R1,R2; /* first decrements R2 and then stores R1 in location whose address is found in R2. */

/*MOV Help*/

Move Register instruction:

Syntax: ***MOV Rx,Ry;***

Example:

MOV R1,R2; /* copies R2 to R1. The value of R2 is retained after the copy. */

/*NMV Help*/

Negate and Move register instruction :

Syntax: ***NMV Rx,Ry;***

Example:

NMV R1,R2; /* copies -R2 to R1. The value of R2 is retained after the copy. */

/*AMV Help*/

Absolute Move Register instruction:

Syntax: ***AMV Rx,Ry;***

Example:

AMV R1,R2; /* copies |R2|, i.e., the absolute value of R2 to R1. The value of R2 is retained after the copy. */

/*IMV Help*/

Increment and Move Register instruction:

Syntax: ***IMV Rx,Ry;***

Example:

IMV R1,R2; /* copies the incremented value of R2 into R1. The value of R2 is not affected by the increment operation. */

/*EXC Help*/

Exchange Register instruction:

Syntax: ***EXC Rx,Ry;***

Example:

EXC R1,R2; /* exchange the contents of R1 and R2. */

/*CLX Help*/

Clear Index Register instruction:

Syntax: ***CLX;***

Example:

CLX; /* Clears the IX register, i.e. , it resets it to 0. */

/*CLR Help*/

Clear Register instruction:

Syntax: ***CLR Rx;***

Example:

CLR R1; /* Clears R1, i.e., it resets to 0. */

/*INC Help*/

Increment Register instruction:

Syntax: ***INC Rx;***

Example:

INC R1; /* increments R1 by 1. If R1 is 5, it becomes 6 after the increment. If it is -5, it becomes -4 after the increment. */

/*DEC Help*/

Decrement Register instruction:

Syntax: ***DEC Rx;***

Example:

DEC R1; /* decrements R1 by 1. If R1 is 5, it becomes 4 after the decrement. If it is -5, it becomes -6 after the decrement. */

/*NEG Help */

Negate Register Instruction:

Syntax: ***NEG Rx;***

Example:

NEG R1; /* negates R1. If R1 is 5, it becomes -5 after negation. If it is -5, it becomes 5 after negation. */

/*CMP Help*/

Complement Register Instruction:

Syntax: ***CMP Rx;***

Example:

CMP R1; /* complements the bits in R1. If R1 is 01110000, it becomes 10001111 after complementation. */

/* SWH Help */

Swap Halves Register instruction:

Syntax: ***SWH Rx;***

Example:

SWH R1; /* swap the two halves of R1 bit by bit. If R1 = 0111 0000, it becomes 00000111 after swapping its halves. */

/*SHF Help */

Shuffle Register instruction:

Syntax: ***SHF Rx;***

Example:

SHF R1; /* shuffles R1 as in shuffling a deck of cards. If R1 = 11110000, it becomes 10101010 after it is shuffled. */

/*REV Help*/

Reverse Register instruction:

Syntax: ***REV Rx;***

Example:

REV R1; /* reverses R1 bit by bit. If R1 = 11000000, it becomes 00000011 after it is reversed. */

/*IOR Help*/

OR instruction:

Syntax: ***IOR Rx,Ry;***

Example:
IOR R1,R2; /* logically ORs the bits of R1 and R2, and stores the result in R1. If R1 is 11110011 and R2 = 11000100, R1 becomes 11110111 after the operation. R2 remains unchanged. */

/*AND Help*/

AND instruction:

Syntax: ***AND Rx,Ry;***

Example:
AND R1,R2; /* logically ANDs the bits of R1 and R2, and stores the result in R1. If R1 is 11110011 and R2 = 11000100, R1 becomes 11000000 after the operation. R2 remains unchanged. */

/*XOR Help*/

Exclusive-OR instruction:

Syntax: ***XOR Rx,Ry;***

Example:
XOR R1,R2; /* logically exclusive-OR the bits of R1 and R2, and stores the result in R1. If R1 is 11110011 and R2 = 11000100, R1 becomes 00110111 after the operation. R2 remains unchanged. */

/*NOR Help*/

Not-OR instruction:

Syntax: ***NOR Rx,Ry;***

Example:
NOR R1,R2; /* computes (R1 OR R2)', and stores the result in R1. If R1 is 11110011 and R2 = 11000100, R1 becomes 00001000 after the operation. R2 remains unchanged. */

/*NND Help*/

Not-AND instruction:

Syntax: ***NND Rx,Ry;***

Example:
NND R1,R2; /* computes (R1 AND R2)', and stores the result in R1. If R1 is 11110011 and R2 = 11000100, R1 becomes 00111111 after the operation. R2 remains unchanged. */

/*XNR Help*/

Exclusive-NOR instruction:

Syntax: XNR Rx,Ry;

Example:
XNR R1,R2; /* exclusive-NOR the bits of R1 and R2, and stores the result in R1. If R1 is 11110011 and R2 = 11000100, R1 becomes 11001000 after the operation. R2 remains unchanged. */

/*ADD Help*/

Add instruction:

Syntax: ***ADD Rx,Ry;***

Example:
ADD R1,R2; /* computes R1 + R2 and stores the result in R1. R2 remains unchanged. */

/*ADC Help*/

Add with Carry instruction:

Syntax: ***ADC Rx,Ry;***

Example:
ADC R1,R2; /* computes R1 + R2 + 1 and stores the result in R1. R2 remains unchanged. */

/*SUB Help*/

Subtract Instruction:

Syntax: ***SUB Rx,Ry;***

Example:
SUB R1,R2; /* computes R1 - R2 and stores the result in R1. R2 remains unchanged. */

/*SUC Help*/

Subtract with Borrow instruction:

Syntax: ***SUC Rx,Ry;***

Example:
SUC R1,R2; /* computes R1 - R2 - 1 and stores the result in R1. R2 remains unchanged. */

/*DAD Help*/

Decimal Add Instruction :

Syntax: ***DAD Rx,Ry;***

Example:
DAD R1,R2; /* computes R1 + R2 in binary-coded-decimal, and stores the result in R1. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*DSU Help*/

Decimal Subtract instruction:

Syntax: ***DSU Rx,Ry;***

Example:
DSU R1,R2; /* computes R1 - R2 in binary-coded-decimal, and stores the result in R1. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*MPY Help*/

Multiply instruction:

Syntax: ***MPY Rx,Ry;***

Example:
MPY R1,R4; /* computes the product of R1 with R2 and stores the result in registers R1 and R2. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*DIV Help*/

Divide   instruction :

Syntax: ***DIV Rx,Ry;***

Example:
DIV R1,R4; /* divides R1 by R4, and stores the quotient in R2, and remainder in R1. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*BTD Help*/

Binary to Decimal conversion instruction:

Syntax: ***BTD Rx;***

Example:
BTD R1; /* converts the binary representation in R1 to BCD. If R1 = 0001 1100 0010 1111, it becomes 0111 0010 0001 1111 after the operation. */

/*DTB Help*/

Decimal to Binary conversion instruction :

Syntax: ***DTB Rx;***

Example:

DTB R1; /* converts the BCD representation in R1 to binary. If R1 = 0111 0010 00011111, it becomes 0001 1100 0010 1111 after the operation. */

/*SHR Help*/

Shift Right register instruction :

Syntax: ***SHR Rx;***

Example:
SHR R1; shifts the bits in R1 right by 1 bit. The leftmost bit is reinserted.
The rightmost bit is lost. If R1 = 11011001,it becomes 11101100 after the shift..

/*SHL Help*/

Shift Left register instruction:

Syntax: ***SHL Rx;***

Example:
SHL R1; /* shifts the bits in R1 left by 1 bit. A 0 bit is inserted on the right. The leftmost bit is lost. If R1 = 11011001, it becomes 10110010 after the shift. */

/*SRC Help*/

Shift Right with Carry instruction :

Syntax: ***SRC Rx;***

Example:
SRC R1; /* shifts R1 right by 1 bit. The carry (C) flag is inserted on the left. The rightmost bit is lost. If C = 0,R1 = 11011001,R1 becomes 01101100 after the shift. */

/*SLC Help*/

Shift left with Carry instruction:

Syntax: ***SLC Rx;***

Example:

SLC R1; /* shifts R1 left by 1 bit. The carry (C) flag is inserted on right. The leftmost bit is lost. If C = 0,R1 = 11011001,R1 becomes 10110010 after the shift. */

/*ROR Help*/

Rotate Right instruction :

Syntax: ***ROR Rx;***

Example:
ROR R1; /* rotates R1 right by 1 bit. The rightmost bit is inserted on the left. If R1 = 11011001, R1 becomes 11101100 after the rotation. */

/*ROL Help*/

Rotate Left instruction :

Syntax: ***ROL Rx;***

Example:
ROL R1; /* rotates R1 left by 1 bit. The leftmost bit is inserted on the right. If R1 = 11011001, R1 becomes 10110011 after the rotation. */

/*RRC Help*/

Rotate Right with Carry instruction:

Syntax: ***RRC Rx;***

Example:
RRC R1; /* rotates R1 right by 1 bit with carry (C) flag . The rightmost bit is placed in the C flag. If C = 0,R1 = 11011001,R1 becomes 01101100, C becomes 1 after the rotation. */

/*RLC Help*/

Rotate Left with Carry instruction:

Syntax: ***RLC Rx;***

Example:
RLC R1; /* rotates R1 left by 1 bit with carry. The leftmost bit is placed in the C flag. If C = 0,R1 = 11011001,R1 becomes 10110010, The carry (C) flag becomes 1 after the rotation. */

/*FPA Help*/

Floating-Point Add instruction:

Syntax: ***FPA Rx,Ry;***

Example:
FPA R1,R3; /* computes the floating-point sum R1 + R3 and stores it in R1. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*FPS Help*/

Floating-Point Subtract instruction:

Syntax: ***FPS Rx,Ry;***

Example:
FPS R1,R3; /* computes the floating-point difference R1-R3 and stores it in R1. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*FPM Help*/

Floating-Point Multiply instruction :

Syntax: ***FPM Rx,Ry;***

Example:
FPM R1,R3; /* computes the floating-point product R1 x R3 and stores it in R1. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*FPD Help*/

Floating-Point Divide instruction:

Syntax: ***FPD Rx,Ry;***

Example:

FPD R1,R3; /* computes the floating-point ratio R1/R3 and stores it in R1. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*CLF Help*/

Clear Frame instruction:

Syntax: ***CLF Frame Address, FrameSize;***

Examples:
CLF 0000,01; /* clears the pixel at the top left corner on the screen . */
LDI R0,1; LDI R1,2; LDI R3,2;
CLF R1R2,02; /* clears the pixels (1,2),(1,3),(2,2),(2,3). */

/*STF Help*/

Set Frame instruction:

Syntax: ***STF Frame Address, FrameSize,Frame Color;***

Examples:
STF 0000,01,03; /* sets the pixel at the top left corner on the screen to red. */
LDI R1,0; LDI R2,3; STF R1R1,01,R2; /* also sets the same pixel to red. */

/*MVF Help*/

Move Frame instruction:

Syntax: ***MVF frame address, frame size, frame address;***

Example:
MVF 0000,01,0201; /* moves the pixel in location (0,0) to location (2,1) on the screen. Larger frames are moved similarly. */

/*CPF Help*/

Copy Frame instruction:

CPF frame address, frame size, frame address;

Example:
CPF 0000,01,0201; /* copies the pixel in location (0,0) to location (2,1) on the screen. Larger frames are copied similarly. */

/*SWF Help*/

Swap Frame instruction:

Syntax: ***SWF frame address, frame size, frame address;***

Example:
SWF 0000,01,0201; /* swaps the pixels in locations (0,0) and (2,1) on the screen. Larger frames are swapped similarly. */

/*FLF Help*/

Flip Frame instruction:

Syntax: ***FLF frame address, frame size, flip orientation;***

Example:
FLF 0000,04,01; /* flips the 4 by 4 frame at the top left corner of the screen about the horizontal line that cuts it into two halves. */

/*ROF Help*/

Rotate Frame instruction:

Syntax: ***ROF frame address, frame size, rotation amount;***

Example:
ROF 0000,04,80; /* rotates a 4 by 4 frame 180 degrees. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*SCF Help*/

Scale Frame instruction :

Syntax:

SCF frame address,frame size,horizontal scale,vertical scale;

Examples:

SCF 0000,02,01,01; /* doubles a 2 by 2 frame. SCF 0000,02,FF,FF; shrinks a 2 by 2 frame by half. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*CHF Help*/

Character Frame instruction:

Syntax: ***CHF frame address,character code,character color;***

Example:
CHF 0000,48,02; /* displays upper case H on the top left corner. Note: CHF instruction works only with 32x32 or higher resolutions. */

/*LDF Help*/

Load Frame instruction:

Syntax: ***LDF frame address,frame size, memory address;***

Example:
LDI R0,X100; LDF 0000,R0,XA; /* loads the set of pixels located in memory locations 000A through 0109 on to the 256x256 screen. */

/*SVF Help*/

Save Frame instruction:

Syntax: ***SVF frame address, frame size, memory address;***

Example:
LDI R0,X100; SVF 0000,R0,XA; /* saves the entire screen into memory locations 000A through 0109. */

/*LSP Help*/

Load Stack Pointer instruction:

Syntax: ***LSP register id or memory address;***

Examples:

LSP R3; /* loads the stack pointer from R3. LSP 14; loads the stack pointer from memory location 14. */

/*SSP Help*/

Save Stack Pointer instruction:

Syntax: ***SSP register id or memory address;***

Examples:
SSP R3; /* saves the stack pointer in R3. SSP 14; save the stack pointer in memory location 14. */

/*CLS Help*/

Clear Stack instruction :

Syntax: ***CLS;***

Example:
CLS; /* clears the stack top, i.e., it resets it to 0. */

/*NES Help*/

Negate Stack instruction:

Syntax: ***NES;***

Example:
NES; /* negates the operand at the top of the stack. */

/*CMS Help*/

Complement Stack instruction :

Syntax: ***CMS;***

Example:
CMS; /* complements the bits of the operand at the top of the stack .*/

/*DUS Help*/

Duplicate Stack instruction:

Syntax: *DUS;*

Example:
DUS; /* duplicates the operand at the top of the stack. It also increments the stack pointer. */

/*SLS Help*/

Shift Left Stack instruction:

Syntax: *SLS;*

Example:
SLS; /* shifts the operand at the top of the stack to left by 1 bit.*/

/*SRS Help*/

Shift Right Stack instruction:

Syntax: *SRS;*

Examples:
SRS; /* shifts the operand at the top of the stack to right by 1 bit. */

/*ADS Help*/

Add Stack instruction:

Syntax: *ADS;*

Example:
ADS; /* adds the two operands at the top two locations of the stack. It also decrements the stack pointer. */

/*SUS Help*/

subtract stack instruction:

Syntax: ***SUS;***

Example:
SUS; /* computes the difference of the two operands at the top two lcations of the stack. It also decrements the stack pointer. */

/*ORS Help*/

OR stack instruction:

Syntax: ***ORS;***

Example:

ORS; /* logically ORs the two operands at the top two locations of the stack. It also decrements the stack pointer. */

/*ANS Help*/

AND stack instruction :

Syntax: ***ANS;***

Example:
ANS; logically ANDs the two operands at the top two locations of the stack. It also
decrements the stack pointer.

/*EXS Help*/

Exchange Stack instruction :

Syntax: ***EXS;***

Example:

EXS; /* exchanges the top two elements at the top two locations of the stack. */

/*PUS Help*/

Pus Stack instruction:

Syntax: ***PUS register- id or memory address;***

Example:
PUS R3; /* pushes the operand in R3 to the top of the stack. It also increments the stack pointer. */

/*POP Help*/

Pop Stack instruction:

POP register id or memory address;

Example:
POP R2; /* pops the operand at the top of the stack to R2. It also decrements the stack pointer. */

/*JMP Help*/

Jump instruction:

Syntax: ***JMP (D or R or T or X), branch address;***

Example:
JMP D,loop; /* forces CodeMill-J to branch to the instruction identified by the label 'loop'. */

/*JCR Help*/

Jump conditional (Relative) instruction. :

Syntax: ***JCR branch condition, relative branch address;***

Example:

JCR R1,<,R2,relative; /* forces CodeMill-J to branch to the instruction that is located at PC + relative if R1 < R2. */

/*JCD Help*/

Jump Conditional (Direct) instruction :

Syntax: ***JCD branch condition, relative branch address;***

Example:
JCD R1,<>,R2,again; /* forces CodeMill-J to branch to the instruction that is located at 'again' if R1 is not equal to R2. */

/*JCT Help*/

Jump Conditional (Indirect) instruction:

Syntax: ***JCT branch condition, relative branch address;***

Example:
JCD R1,=,0,indirect; /* forces CodeMill-J to branch to the address that is specified in the location 'indirect' if R1 = 0. */

/*JCX Help*/

Jump conditional (Index) instruction:

Syntax: ***JCX branch condition, index displacement;***

Example:
JCX R3,=,1,4; /* forces CodeMill-J to branch to the address that is specified in IX if R3 = 1, and then increments IX by 4. */

/*JSR Help*/

Subroutine Call instruction :

Syntax: ***JSR subroutine address;***
or
JSR subroutine address, R;

Examples:
JSR subroutine1; /* forces CodeMill-J to jump to the subroutine identified by the label 'subroutine1'. */
JSR subroutine1; /* same as above plus, the scratchpad registers are saved before CodeMill Jr. branches to the subroutine. */

/*RTS Help*/

Return From Subroutine instruction:

Syntax: RTS;

Example:

JSR subr; HLT;
subr: ADD R1,R2; RTS; /* forces CodeMill-J to return to HLT instruction after it executes the ADD instruction in the subroutine. */

/*NOP Help*/

No-operation instruction:

Syntax: *NOP operand;*

Example:
NOP 10; /* forces CodeMill-J to execute 10 cycles without performing any operation. NOP can be used to introduce delay between excutions of instructions. */

/*SCC Help*/

Set Condition Codes instruction:

Syntax: *SCC Register or numerical operand;*

Examples:
SCC 7; /* complements the overflow and sign flags SCC 5; sets the overflow and sign flags. Refer to Introduction To Computers Using CodeMill-J for more information. */

/*HLT Help*/

Halt instruction:

Syntax: *HLT;*

Example:
HLT; /* halts the execution of a program upon the completion of its execution. */

/*SND Help*/

Send instruction :

Syntax: ***SND Register id,device id/data;***

Example:
LDI R1,0; SND R1,3; /* sends a red pixel to the screen. */

/*RCV Help*/

Receive instruction:

Syntax: ***RCV Register id,device id;***

Example:
RCV R1,0; /* forces CodeMill-J to receive a character from the keyboard. */

8.2 Command Keys

/*STEP Help*/

STEP forces CodeMill-J to execute the instruction in the instruction register.

/*EXEC Help*/

EXEC forces CodeMill-J to execute the program in the text area of this window.

/*SUSP Help*/

SUSP forces CodeMill-J to suspend executing the program in the text area of this window.

/*RESM Help*/

RESM forces CodeMill-J will resume executing the program in the text area of this window.

/*RAM Help*/

RAM brings up or hides the RAM memory.

/*VRAM Help*/

VRAM brings up or hides the VRAM memory.

/*STACK Help*/

STCK brings up or hides the STACK memory.

/* CPUR Help*/

CPUR brings up or hides the scratchpad and control register.

/* RSTA Help*/

RSTA resets all the registers and clears the screen.

/* CSRC Help*/

CSRC clears the picture on the screen.

/* RSPC Help*/

RSPC resets the program counter.

/* RSSP Help*/

RSSP resets the stack pointer.

/* RSIX Help*/

RSIX resets the IX register.

/* H/SCN Help*/

H/SCN toggles the background scene on the screen in and out view.

/* H/SRN Help*/

H/SRN toggles the background grid on the screen in and out view.

/* H/SPD Help*/

H/SPD toggles the sratchpad registers in and out of view.

/* BNRY Help*/

BNRY sets the registers to binary.

/* OCTL Help*/

OCTL sets the registers to octal.

/* DCML Help*/

DCML sets the registers to decimal.

/* XCML Help*/

XCML sets the registers to hexadecimal.

/* SPSZ+ Help*/

SPSZ+ sets the scratchpad size to the next value in the menu.

/* SPSZ- Help*/

SPSZ-   sets the scratchpad size to the previous value in the menu.

/* RMSZ+ Help*/

RMSZ+ sets the RAM size to the next value in the menu.

/* RMSZ- Help*/

RMSZ- sets the RAM size to the previous value in the menu.

/* VRSZ+ Help*/

VRSZ+ sets the VRAM size to the next value in the menu.

/* VRSZ- Help*/

VRSZ- sets the VRAM size to the previous value in the menu.

/* STSZ+ Help*/

STSZ+ sets the STACK size to the next value in the menu.

/* STSZ- Help*/

STSZ- sets the STACK size to the previous value in the menu.

/* RESL+ Help*/

RESL+ sets the GRID Resolution to the next value in the menu.

/* RESL- Help*/

RESL- sets the GRID Resolution to the previous value in the menu.

/* SCRN+ Help*/

SCRN+ sets the Screen size to the next value in the menu.

/* SCRN- Help*/

SCRN- sets the screen size to the previous value in the menu.

/* SCN+ Help*/

SCRN+ sets the scene to the next value in the menu.

/* SCN- Help*/

SCRN- sets the scene to the previous value in the menu.

/* COLR+ Help*/

COLR+ sets the Color to the next value in the menu.

/* COLR- Help*/

COLR- sets the Color to the previous value in the menu.

9. CodeMill-J Help on the Web

CodeMill-J is a dynamic learning technology. To keep up with the most up-to-date information about CodeMill-J, please visit www.sabatech.com often.